Assignment vLast: Where's the file?

Sam Olagun Maanya Tandon

Make Targets	Description
`make all`	Builds the server, client, and tests and puts them into separate folders.
`make clean`	Removes server, client, and test folders.
`make client`	Builds the client, `wtf_client.c`, and moves it to the `client` folder.
`make server`	Builds the server, `wtf_server.c`, and moves it to the `server` folder.
`make test`	Builds the tests, `wtf_test.c`, and moves them to the `test` folder.
`make run_server`	Shortcut for running the server on port :8000.

Command	Description
`add`	Adds a `<file>` to the client *.Manifest* for a specific project.
`checkout`	Downloads a `<project>` to the client from the server if it exists.
`commit`	Stores client changes for `<project>` to a `.Commit` file stored on both the server and the client.
`create`	Creates a new `<project>` on the server.
`currentversion`	Lists the current version of every file in `<project>`.
`destroy`	Removes `<project>` from the server.
`history`	Lists the version of every past file in `<project>`.
`push`	Pushes and applies valid changes listed in `.Commit`for `<project>` to the server.
`remove`	Removes a `<file>` from the client *.Manifest* for a specific project.
`rollback`	Deletes all history items after `<version>` for `<project>` and reverts to `<version>`.
`update`	Stores all server changes for `<project>` an `.Update` file stored on the client.
`upgrade`	Downloads and applies changes in `.Update` for `<project>` to the client.

Command	Usage
`add`	`./WTF add <project> <file>`
`checkout`	`./WTF checkout <project>`
`commit`	`./WTF commit <project>`
`create`	`./WTF create <project>`
`currentversion`	`./WTF currentversion <project>`
`destroy`	`./WTF destroy <project>`
`history`	`./WTF history <project>`
`push`	`./WTF push <project>`
`remove`	`./WTF remove <project> <file>`
`rollback`	`./WTF rollback <project> <version>`
`update`	`./WTF update <project>`
`upgrade`	`./WTF upgrade <project>`

Dot Files

`.Manifest`

Our `.Manifest` file follows the assignment description exactly.

```
<project_version>
<file_path> <file_version> <file_hash>
```

`.Commit`, `.Update` and `.Conlfict`

Our `.Commit`, `.Update` and `.Conflict` files follow the assignment description exactly as well.

```
A <file_path> <hash>
M <file_path> <hash>
D <file_path> <hash>
```

Modularity and Program Design

We knew that our project was going to get large quickly, so we organized it into modules. We created an `includes` folder containing interfaces for every function created and a `util` folder for commonly used utilities (See: *src* for all of the modules that we created and *src/util* for all of the utilities). We also separated commands into C files (See: *src/commands*) and created make shortcuts that made compiling and unit testing easy (See: *Makefile*).

Compression

Our project implements all 3 parts of the extra credit. We compress all files that are sent between the server and client (See: *src/request.c:63*, *src/request.c:29*, *src/response.c:31*, *src/response.c:63*) and compress all items pushed to history (See: *src/commands/push.c:148*) and we do this all without system calls (See: *src/compression.c*). A big part of this was modularity—making the tools that we needed to store files in memory easily (See: *src/filelist.c*) and abstracting away compression (See: *src/compression.c*).

Threading and Mutexes

We kept our threading code as simple as possible. We launch a new thread on every connection that is recieved (See: *src/server.c:28*). Our threads only last the lifetime of the program. We also kept our mutex code as simple as possible. We stored our mutexes in a linked list called `MutexList`.

```
struct MutexList {
  char *project_name;
  bool is_locked;
  pthread_mutex_t *lock;
  struct MutexList *next;
};
```

The linked list stores essential information about each mutex inlcuding it's lock status. We created functions `add_project`, `remove_project`, `lock_project`, and `unlock_project` for performing mutex operations based on project names (See: *src/mutexlist.c*). These functions were complete with error checking so that you couldn't lock a unlock twice and such operations would be logged as errors (See: *src/mutexlist.c:123*). We

also return booleans on each mutex operation to indicate success or failure. This is important because mutex operations aren't guarenteed to be legal, so it's important that that mutex functions communicate bad behavior to their callers.

We call our mutex functions in 3 places. For all commands that aren't create or destory, we call a mutex on the project that the command involves (See: *wtf_server.c:48*). For create, we create a new mutex and lock it immediately (See: *create.c:66*). We unlock that mutex before we return to the client (See: *create.c:78*, *create.c:107*). For destroy, we surround the critical section with lock and unlock, then unlock and remove the project mutex once we're done. The removed mutex may still have users, but the destroy has already occured at this point and the project no longer exists, so this is not a problem.

Protocol

C's standard socket library is very primitive. It makes sending bytes with a known length simple, but makes sending files and data fields more difficult. Because this assignment required us to send both files and data fields, we created a protocol for communicating this type of information.

Our "protocol" is the `Request` grammar and the `Response` grammar. These grammars specify how each `Request` and `Response` sent will appear in memory. Before I describe how we implemented `Request` and `Response`, here are the grammars for each. We arrived at these two grammars by modifying the recommended protocol in the assignment description.

Request

```
<message> ":" <status_code> ":"
<command_name> ":" <project_name> ":"
<project_version> ":" <file_count> ":"
<file_path> ":" <file_version> ":" <file_hash> ":" <file_size> ":" <file_bytes>
```

Response

```
<message> ":" <status_code> ":"
<project_version> ":" <file_count> ":"
<file_path> ":" <file_version> ":" <file_hash> ":" <file_size> ":" <file_bytes>
```

To implement our protocol, we wrote *write* and *read* functions that wrote a `Request`/`Response` to a file descriptor and read a `Request`/`Response` from a socket file descriptor respectively. We also created `Request` and `Response` structures to match our protocol.

```
// Taken from `includes/src/request.h`.
struct Request {
  char* message;
  int status_code;
  char* command_name;
  char* project_name;
  int project_version;
  int file_count;
  FileList* filelist;
};

Request* request_read(int fd);
void request_write(int fd, Request* request);
```

```c
// Taken from `includes/src/response.h`.
struct Response {
  char* message;
  int status_code;
  int project_version;
  int file_count;
  FileList* filelist;
};

Response* response_read(int fd);
void response_write(int fd, Response* response);
```

Writing code this way allowed us to focus more on what we were sending and less on how we were sending it. It also made debugging easier because we created `request_log` and `response_log` functions to log reads and writes.

Miscellaneous

Reading files into memory

The only files read are those listed in `.Manifest`. To send files, read in the manifest, then read file data into every file listed in the manifest.

```c
// Read in the manifest
Manifest* manifest = manifest_read("projects/test_project");

// Get files listed in the manifest
// Each file contains `file_name`, `file_version` and `file_hash`. See `filelist.h`
FileList* manifest_files = manifest->filelist;

// Read file data into every listed file
// Each file now now also contains `file_size` and `file_bytes`. See `filelist.h`
FileList* files_with_data = filelist_readbytes("projects/test_project", manifest_files);
```

Sending files with `Response` and `Request`

After successfuly storing files inside of a `FileList` linked list, add them to a request to send them. Open includes/src/request.h to see what else that a request can contain.

```c
// Read in manifest
Manifest* manifest = manifest_read("projects/test_project");

// Get files listed in the manifest
FileList* manifest_files = manifest->filelist;

// Read file data into every listed file
FileList* files_with_data = filelist_readbytes("projects/test_project", manifest_files);

// Create request
Request* request = request_new();

// Send files with request
request->filelist = files_with_data;
```

```
Response* response = client_send(request);
```